

Caches

Reminders:

Lab 5 due today

Quiz 2, Tuesday Nov 10th, 7:30-9:30pm

Covers L8-L16

Quiz 2 review, Monday Nov 9th 7:30-9:30pm

Practice quizzes should be released by Saturday

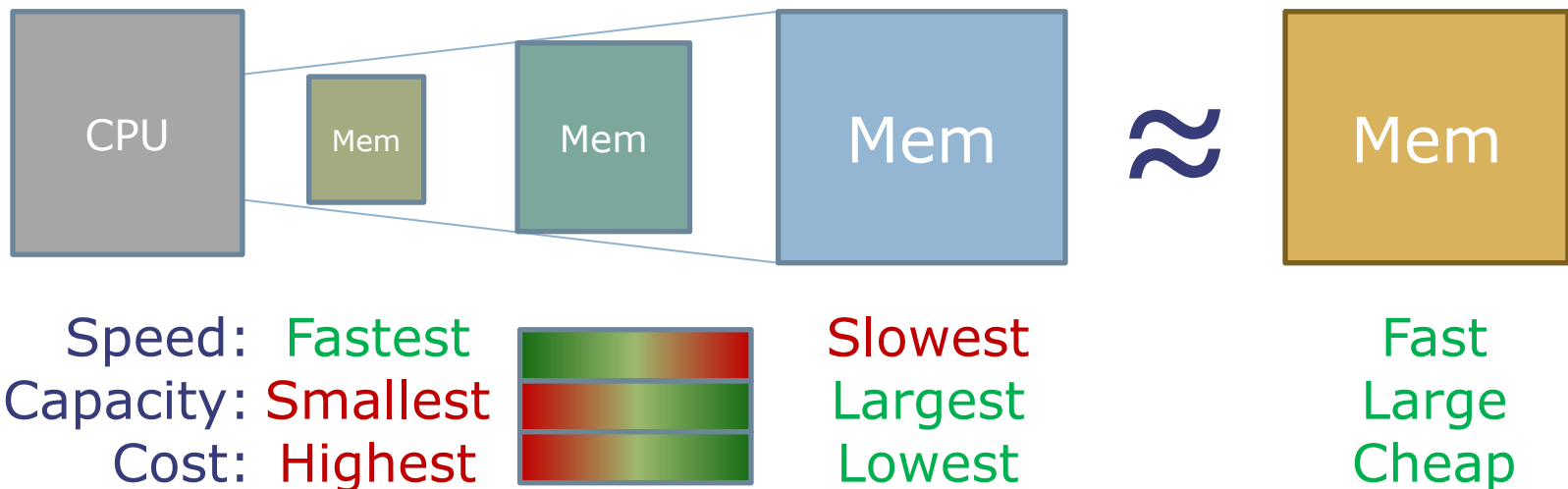
The Memory Hierarchy

Want large, fast, and cheap memory, but...

Large memories are slow (e.g., Hard Disk)

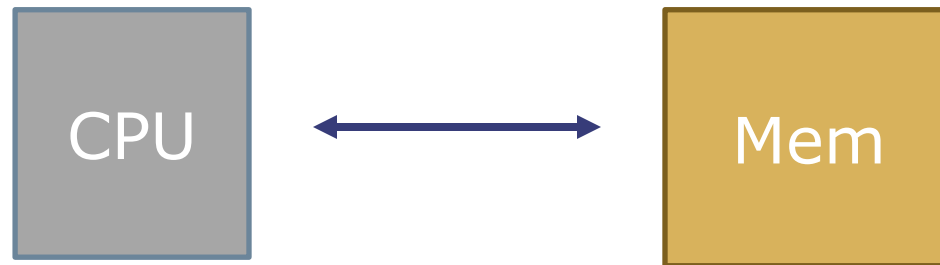
Fast memories are small and expensive (e.g., SRAM)

Solution: Use a **hierarchy** of memories with different tradeoffs to **fake** a large, fast, cheap memory



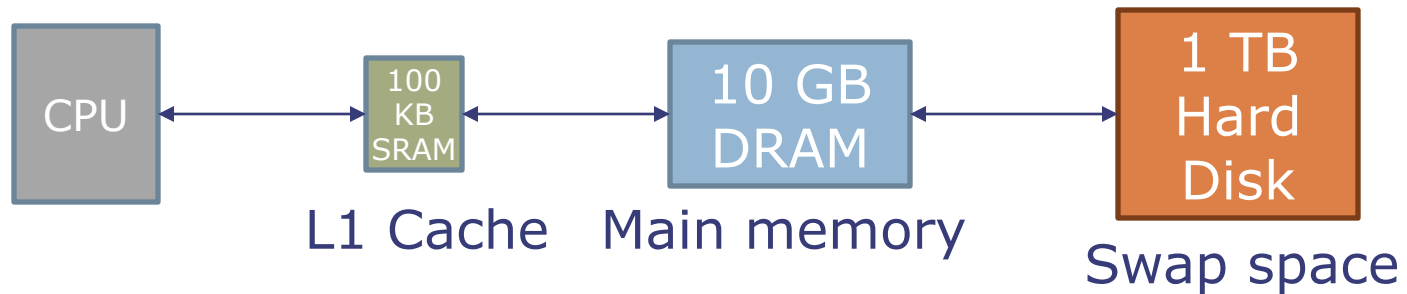
Memory Hierarchy Interface

- Programming model: Single memory, single address space



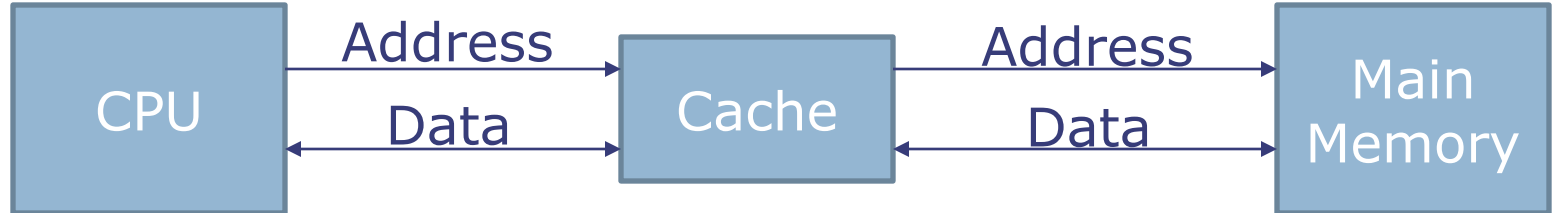
Memory Hierarchy Interface

- Programming model: Single memory, single address space
- Machine transparently stores data in fast or slow memory, depending on usage patterns



Caches

- Cache: A small, interim storage component that transparently retains (caches) data from recently accessed locations



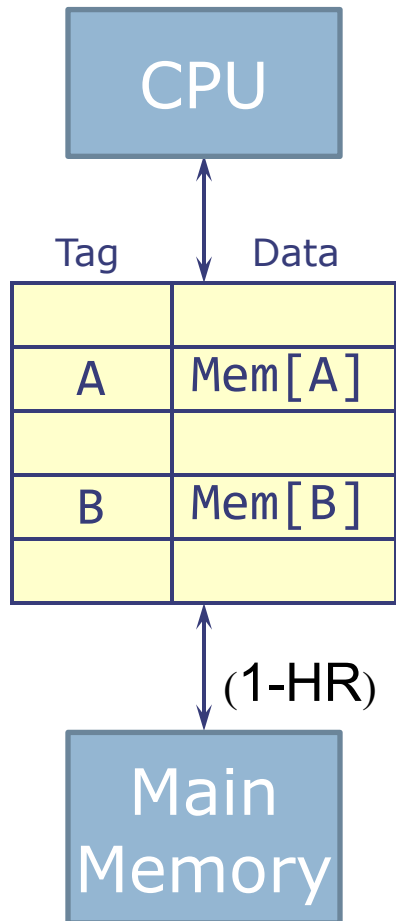
- Processor sends accesses to cache. Two options:
 - Cache hit**: Data for this address in cache, returned quickly
 - Cache miss**: Data not in cache
 - Fetch data from memory, send it back to processor
 - Retain this data in the cache (replacing some other data)
 - Processor must deal with variable memory access time

Why Caches Work

- Two predictable properties of memory accesses:
 - **Temporal locality**: If a location has been accessed recently, it is likely to be accessed (reused) soon
 - **Spatial locality**: If a location has been accessed recently, it is likely that nearby locations will be accessed soon
- Result:
 - High hit rate (low miss ratio)
 - Reduced Average Memory Access Time (AMAT):

$$AMAT = HitTime + MissRatio \times MissPenalty$$

Basic Cache Algorithm (Reads)



On reference to Mem[X],
look for X among cache tags

HIT: $X = \text{Tag}(i)$
for some
cache line i

Return Data(i)

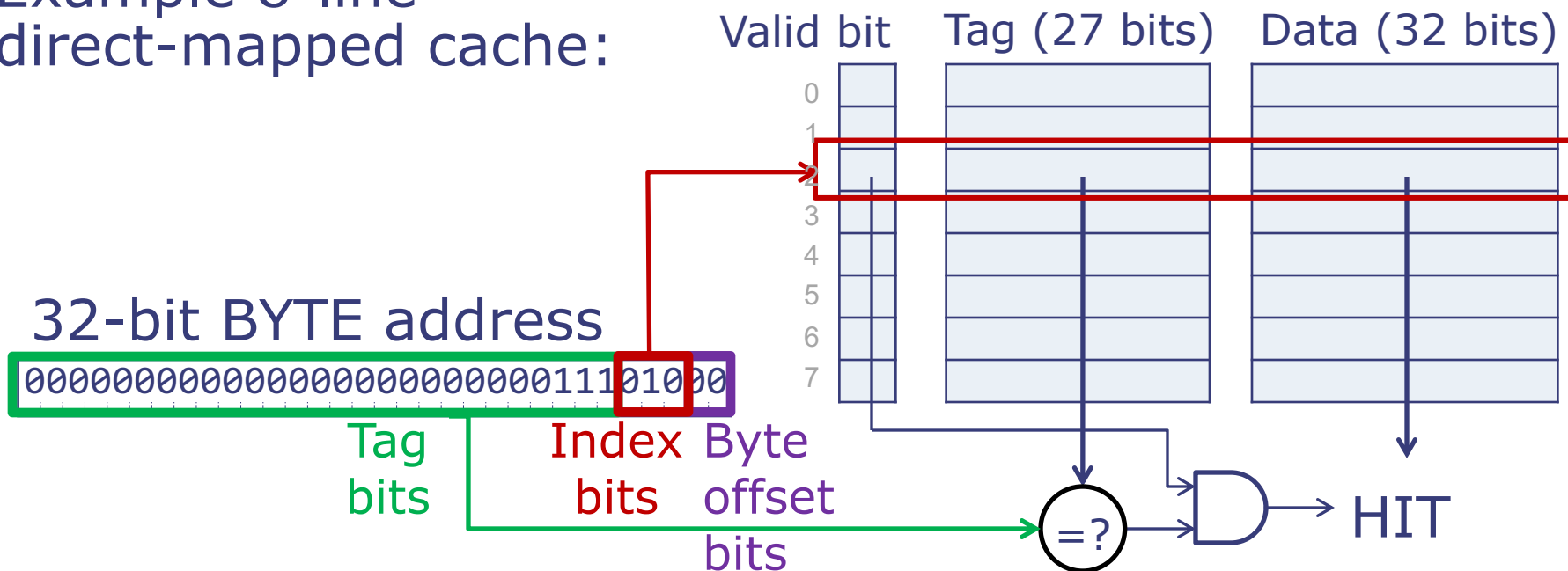
MISS: X not
found in Tag
of any cache line

Read Mem[X]
Return Mem[X]
Select a line k
to hold Mem[X]
Write Tag(k) = X,
Data(k) = Mem[X]

Q: How do we "search" the cache?

Direct-Mapped Caches

- Each word in memory maps into a single cache line
- Access (for cache with 2^W lines):
 - Index into cache with W address bits (the **index bits**)
 - Read out valid bit, tag, and data
 - If valid bit == 1 and tag matches upper address bits, HIT
- Example 8-line direct-mapped cache:



Example: Direct-Mapped Caches

64-line direct-mapped cache → 64 indices → 6 index bits

Read Mem[0x400C]

0100 0000 0000 1100
TAG: 0x40
INDEX: 0x3
BYTE OFFSET: 0x0

HIT, DATA 0x42424242

Would 0x4008 hit?

INDEX: 0x2 → tag mismatch
→ MISS

	Valid bit	Tag (24 bits)	Data (32 bits)
0	1	0x000058	0xDEADBEEF
1	1	0x000058	0x00000000
2	1	0x000058	0x00000007
3	1	0x000040	0x42424242
4	0	0x000007	0x6FBA2381
	⋮	⋮	⋮
63	1	0x000058	0xF7324A32

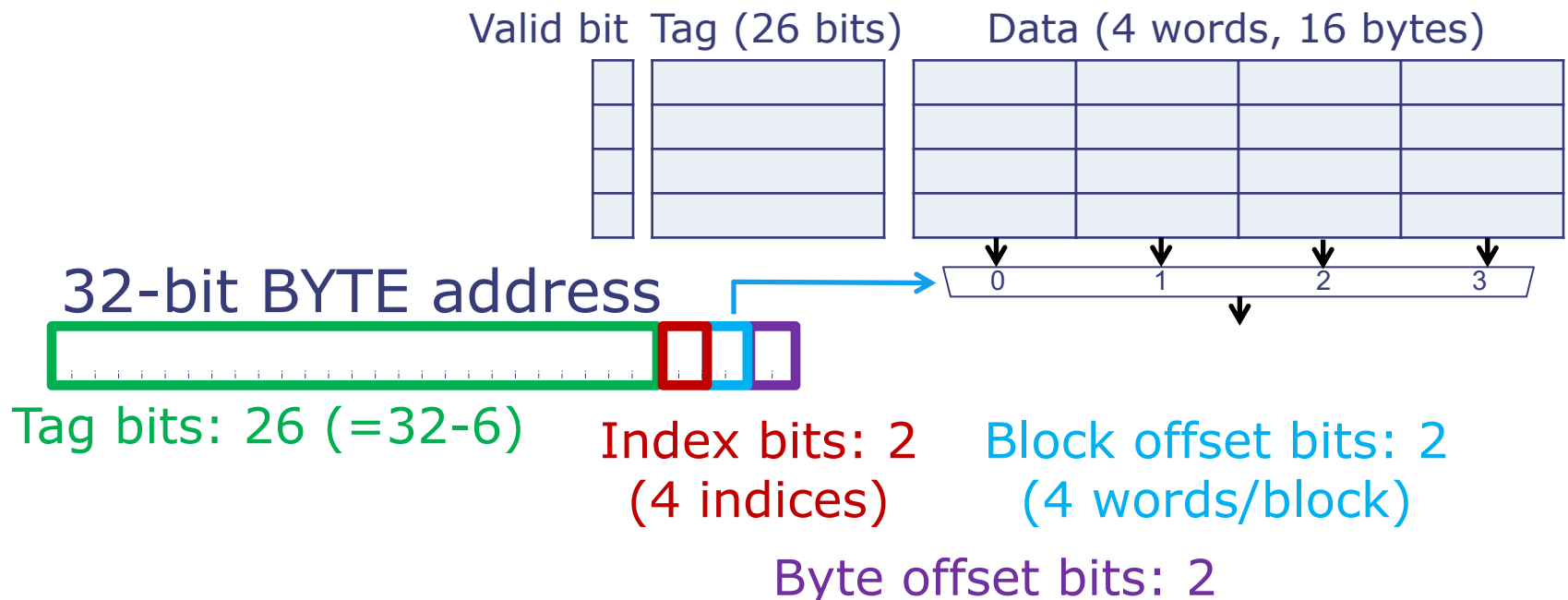
Part of the address (index bits) is encoded in the location
Tag + Index bits unambiguously identify the data's address

Selection of Index Bits

- Why do we chose low order bits for index?
 - Allows consecutive memory locations to live in the cache simultaneously
 - Reduces likelihood of replacing data that may be accessed again in the near future
 - Helps take advantage of locality

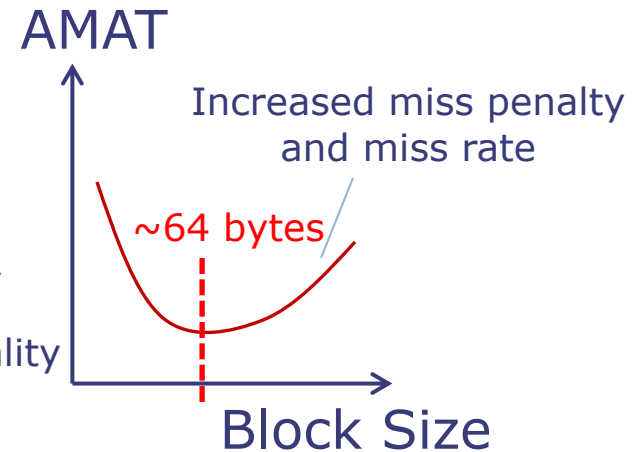
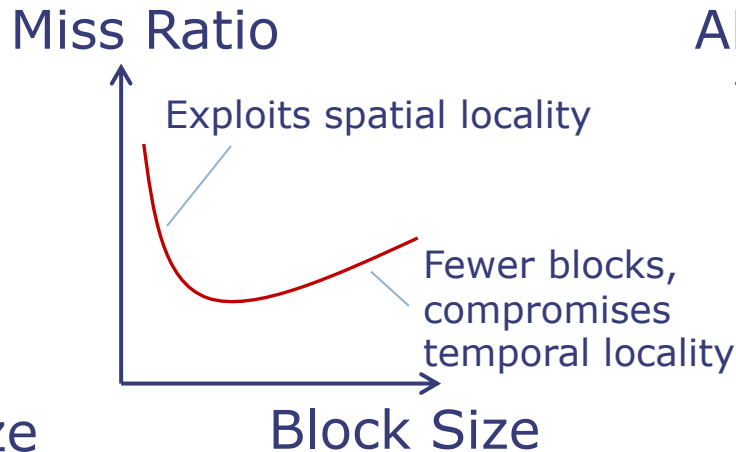
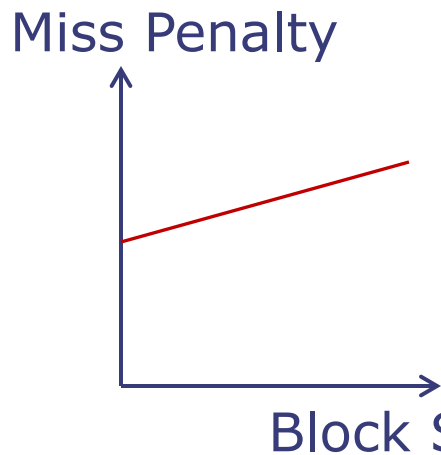
Block Size

- Take advantage of spatial locality: Store multiple words per data line
 - Always fetch entire block (multiple words) from memory
 - Another advantage: Reduces size of tag memory!
 - Potential disadvantage: Fewer indices in the cache
- Example: 4-block, 16-word direct-mapped cache



Block Size Tradeoffs

- Larger block sizes...
 - Take advantage of spatial locality
 - Incur larger miss penalty since it takes longer to transfer the block from memory
 - Can increase the average hit time and miss ratio
- $AMAT = HitTime + MissPenalty * MissRatio$



Direct-Mapped Cache Problem: Conflict Misses

Word Address	Cache Line index	Hit/ Miss
-----------------	---------------------	--------------

Loop A:
Code at
1024,
data at
37

1024	0	HIT
37	37	HIT
1025	1	HIT
38	38	HIT
1026	2	HIT
39	39	HIT
1024	0	HIT
37	37	HIT
...		

Assume:

1024-line DM cache

Block size = 1 word

Consider looping code, in
steady state

Assume WORD, not BYTE,
addressing

Loop B:
Code at
1024,
data at
2048

1024	0	MISS
2048	0	MISS
1025	1	MISS
2049	1	MISS
1026	2	MISS
2050	2	MISS
1024	0	MISS
2048	0	MISS
...		

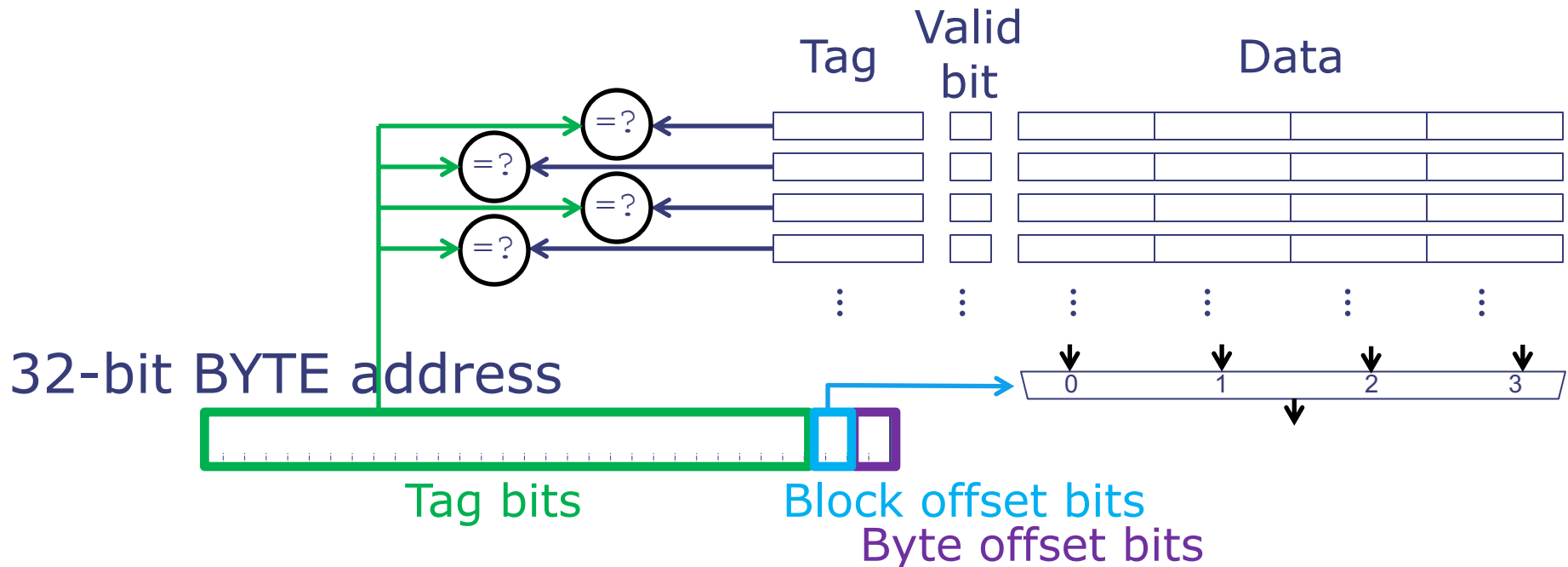
Inflexible mapping
(each address can only be
in one cache location) →

**Conflict misses (multiple
addresses map to same
cache index)!**

Fully-Associative Cache

Opposite extreme: Any address can be in any location

- No cache index!
- **Flexible** (no conflict misses)
- **Expensive**: Must compare tags of all entries in parallel to find matching one

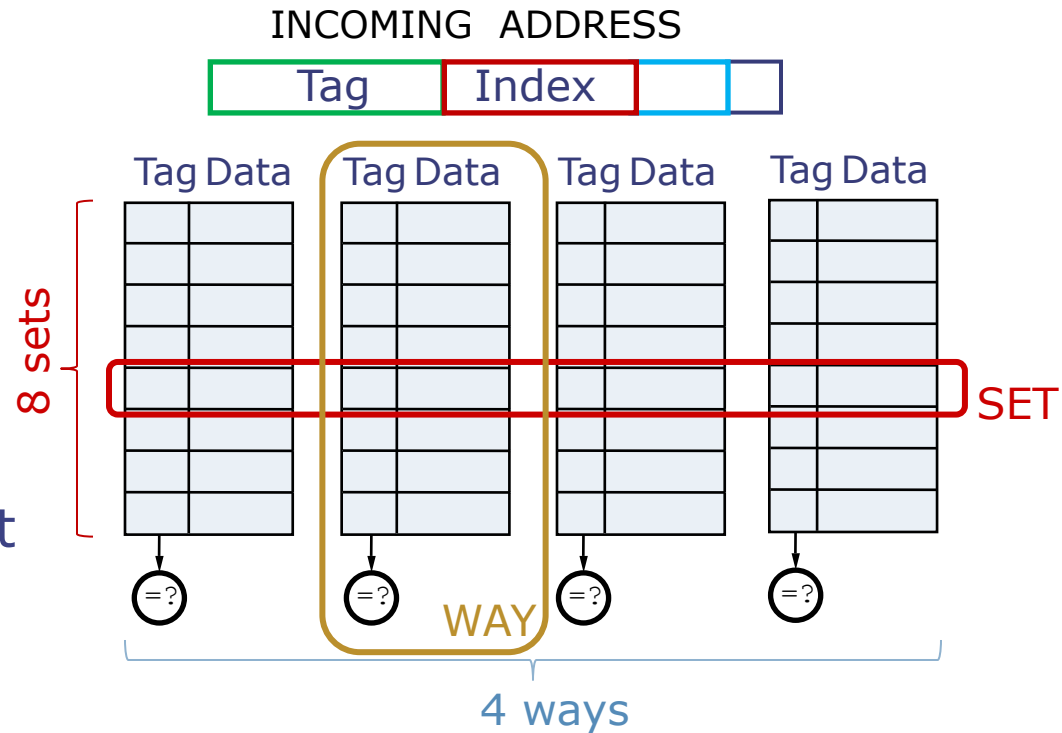


N-way Set-Associative Cache

- Use multiple direct-mapped caches in parallel to reduce conflict misses

- Nomenclature:

- # Rows = # Sets
- # Columns = # Ways
- Set size = #ways
= "set associativity"
(e.g., 4-way → 4 lines/set)
- Each address maps to only one set, but can be in any way within the set
- Tags from all ways are checked in parallel

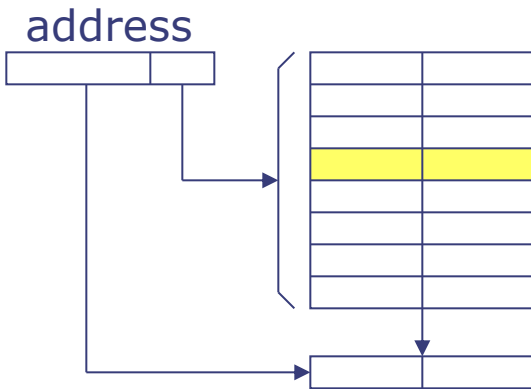


- Fully-associative cache: Extreme case with a single set and as many ways as cache lines

Associativity Implies Choices

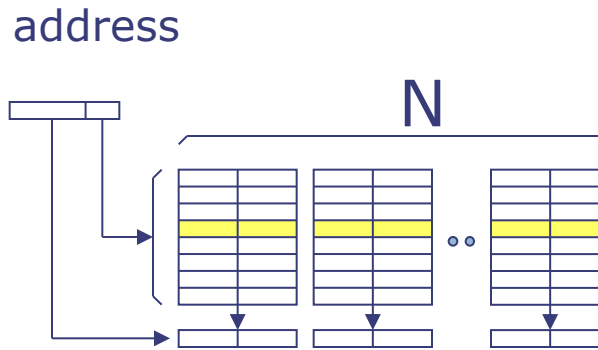
Issue: Replacement Policy

Direct-mapped



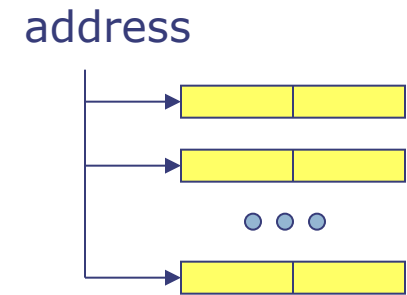
- Compare addr with only one tag
- Location A can be stored in exactly one cache line

N-way set-associative



- Compare addr with N tags simultaneously
- Location A can be stored in exactly one set, but in any of the N cache lines belonging to that set

Fully associative



- Compare addr with each tag simultaneously
- Location A can be stored in any cache line

Replacement Policies

- Optimal policy: Replace the line that is accessed furthest in the future
 - Requires knowing the future...
- Idea: Predict the future from looking at the past
 - If a line has not been used recently, it's often less likely to be accessed in the near future (a locality argument)
- **Least Recently Used (LRU)**: Replace the line that was accessed furthest in the past
 - Works well in practice
 - Need to keep ordered list of N items $\rightarrow N!$ orderings $\rightarrow O(\log_2 N!) = O(N \log_2 N)$ "LRU bits" + complex logic
 - Caches often implement cheaper approximations of LRU
- Other policies:
 - First-In, First-Out (least recently replaced)
 - Random: Choose a candidate at random
 - Not very good, but does not have adversarial access patterns

Write Policy

Write-through: CPU writes are cached, but also written to main memory immediately (stalling the CPU until write is completed). Memory always holds current contents

- Simple, slow, wastes bandwidth

Write-back: CPU writes are cached, but not written to main memory until we replace the line. Memory contents can be “stale”

- Fast, low bandwidth, more complex
- Commonly implemented in current systems

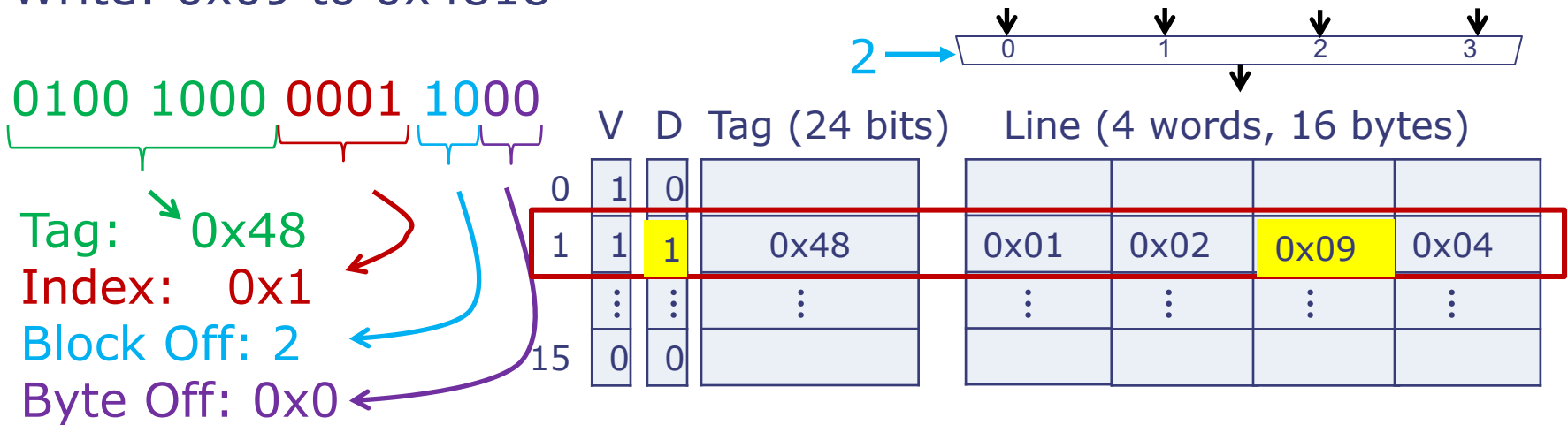
Example: Cache Write-Hit

16-line direct-mapped cache \rightarrow 4 index bits

Block size = 4 \rightarrow 2 block offset bits

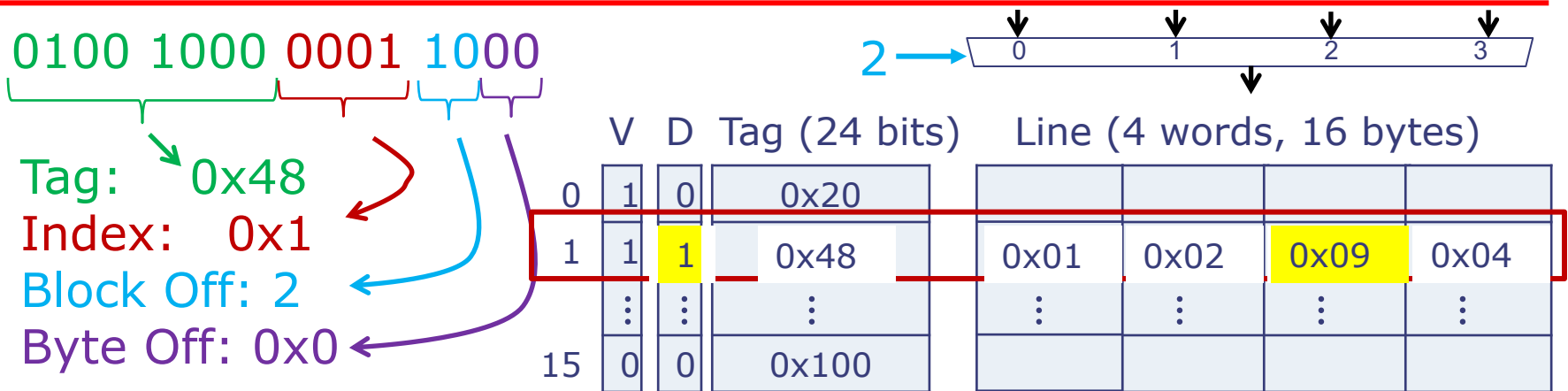
Write Policy = Write Back

Write: 0x09 to 0x4818



D=1: cache contents no longer match main memory so write back line to memory upon replacement

Example: Cache Write-Miss



Write: 0x09 to 0x4818

1. Tags don't match -> Miss

- **D=1**: Write cache line 1 (tag = 0x280: addresses 0x28010-0x2801C) back to memory
- If D=0: Don't need to write line back to memory.

2. Load line (tag = 0x48: addresses 0x4810-0x481C) from memory

3. Write 0x09 to 0x4818 (block offset 2), set D=1.

Summary: Cache Tradeoffs

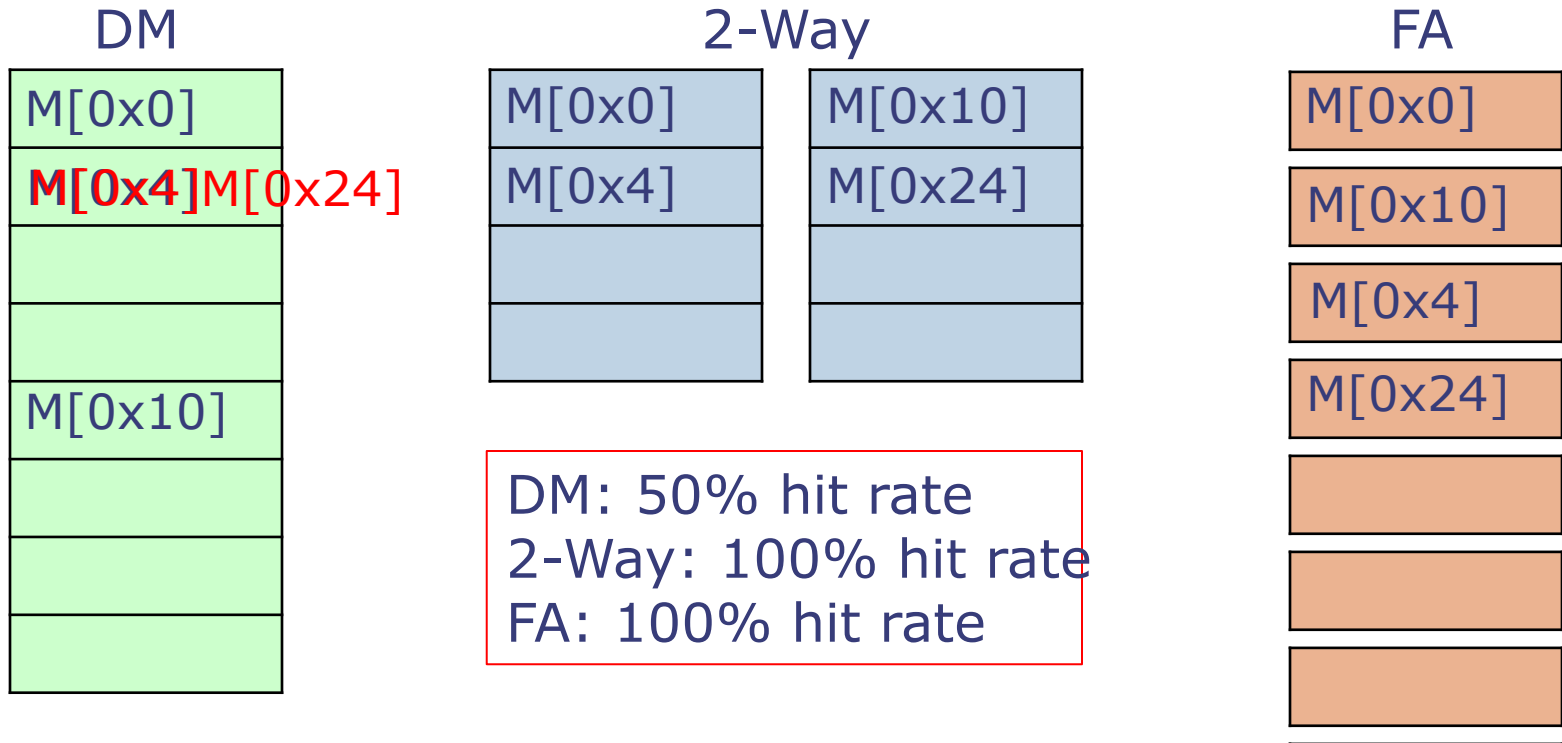
$$AMAT = HitTime + MissRatio \times MissPenalty$$

- Cache size
- Block size
- Associativity
- Replacement policy
- Write policy

Example: Comparing Hit Rates

3 Caches: DM, 2-Way, FA: each has 8 words, block size=1, LRU

Access following addresses repeatedly: 0x0, 0x10, 0x4, 0x24



0x0 = 0b000000

DM index = 000

2-Way index = 00

0x10 = 0b010000

DM index = 100

2-Way index = 01

0x4 = 0b000100

DM index = 001

2-Way index = 01

0x24 = 0b100100

DM index = 001

2-Way index = 01

Example 2: Comparing Hit Rates

Access: 0x0, 0x4, 0x8, 0xC, 0x10, 0x14, 0x18, 0x1C, 0x20 repeatedly

DM

M[0x0]
M[0x20]
M[0x4]
M[0x8]
M[0xC]
M[0x10]
M[0x14]
M[0x18]
M[0x1C]

2-Way

M[0x0]	M[0x10]
M[0x20]	M[0x0]
M[0x10]	M[0x20]
M[0x4]	M[0x14]
M[0x8]	M[0x18]
M[0xC]	M[0x1C]

FA

M[0x0] M[0x20]
M[0x4] M[0x0]
M[0x8] M[0x4]
M[0xC] M[0x8]
M[0x10] M[0xC]
M[0x14] M[0x10]
M[0x18] M[0x14]
M[0x1C] M[0x18]

DM: Hit rate = 7/9

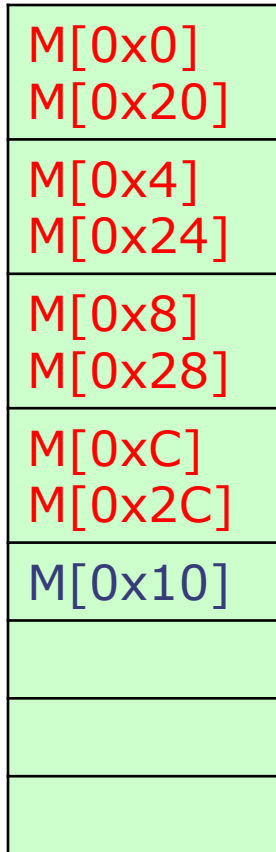
2-Way: Hit rate = 6/9

FA: Hit rate = 0%

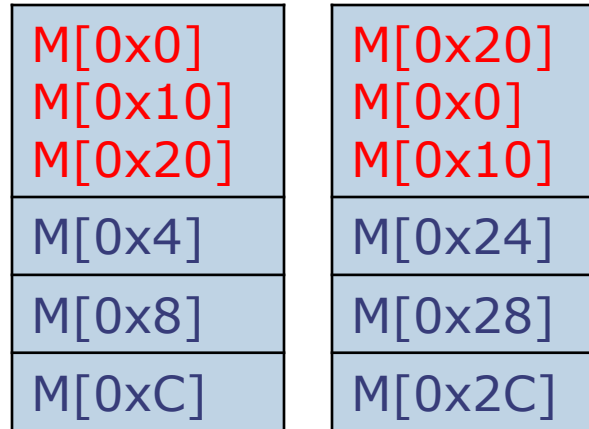
Example 3: Comparing Hit Rates

Access: 0x0, 0x4, 0x8, 0xC, 0x20, 0x24, 0x28, 0x2C, 0x10
repeatedly

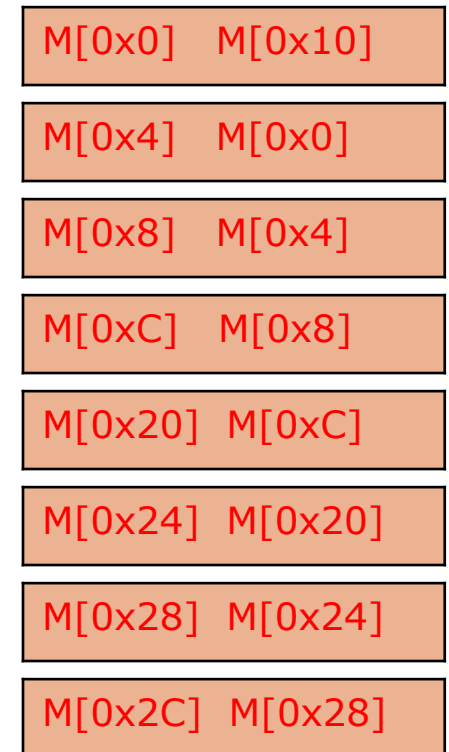
DM



2-Way



FA



DM: Hit rate = 1/9

2-Way: Hit rate = 6/9

FA: Hit rate = 0%

Thank you!

*Next lecture: Operating
Systems*